Northeastern University - Seattle

Khoury College of Computer Sciences

CS6200 Information Retrieval Fall 2019

Lecture 5: Indexing and Ranking Data

Sep 30, 2019

Administrivia

- Assignment 1
  - Feedback?
- Quiz 2 evaluated. Check it out.

Quiz 3 Discussion .. 1

Good answers overall!

1. Zipf's Law : r.Pr = c . Assumption: Corpus in English, so c ~ 0.1.

Pr = 6391/ 572219

Rank r = 0.1 / (6391/572219) = 8.95 or approximately 9

Some of you approximated Pr to 0.01. Why? So you got the answer as 10. OK for now, but why approximate here? One of you got 1??

Also, some of you left the rank at 8.95 (or 8.93). That's not a complete answer.

Simulating a collection is not a great idea. Especially when you have Zipf's Law.

2.. Stopwords for Wikipedia Pages could include:

Wiki, Wikipedia, edit, "See also", disambiguation, …

Note: I wanted stopwords specific to Wikipedia. So of, the, span, utc, etc. are not good answers

Quiz 3 Discussion ..2

3. Dealing with multi-lingual docs.
Unlimited resources: Use Machine Translation/keep track of each language para/segment, apply appropriate transforms.
Limited resources: Look for dominant language in doc/query/settings, use that. Or while querying, the user could specify the specific language they want their results from, and a higher priority could be assigned for documents labelled with the specified language.

Very good range of answers.

But one of you sent mail at 11pm Sunday, asking for a clarification.
Was there a serious expectation of an answer?

4. TREC

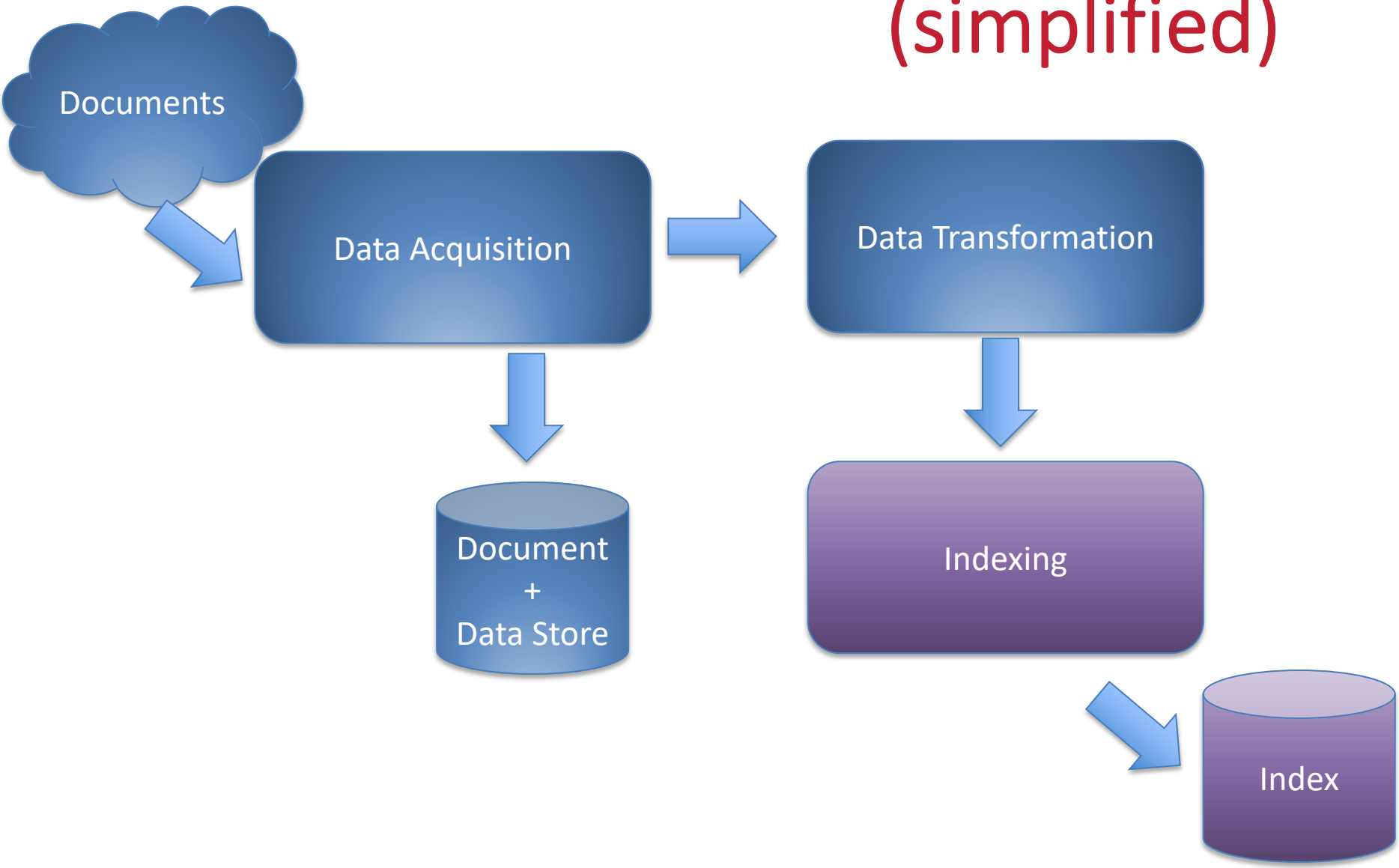
For example, the decision track, with the goal of improving decision-making based on retrieval.

Good responses, thanks!

Overview

- Indexes & Ranking
  - Much more about ranking in later lectures
- Inverted Indexes
- Compressing Inverted Indexes
- Constructing Indexes

Search Engine: Behind the Scenes (simplified)



INDEXES & RANKING

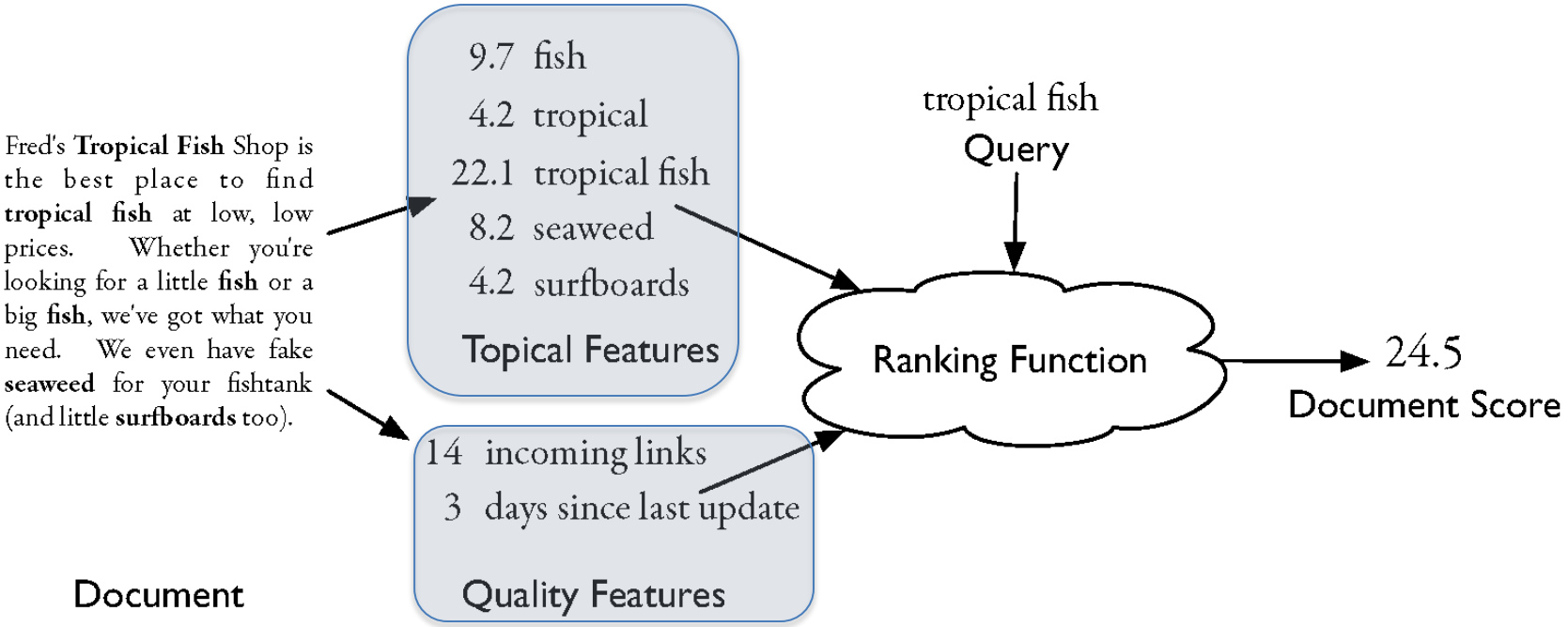
What are Indexes?

- *Indexes* are data structures designed to make search faster
  - e.g. Telephone directory
- Text search has unique requirements, which leads to unique data structures
- Most common data structure: *inverted index*
  - general name for a class of structures
  - "inverted" because documents are associated with words, rather than words with documents
    - similar to a *concordance*
      e.g. http://www.opensourceshakespeare.org/concordance/

Indexes and Ranking

- Indexes are designed to support *search*
  - faster response time, supports updates
- Text search engines use a specific type of search: *ranking search*
  - documents are retrieved in sorted order,
    according to a score computed
    using the document representation,
    the query, and a *ranking algorithm*
- What is a reasonable abstract model for ranking?
  - can discuss indexes without details of retrieval model

Abstract Model of Ranking

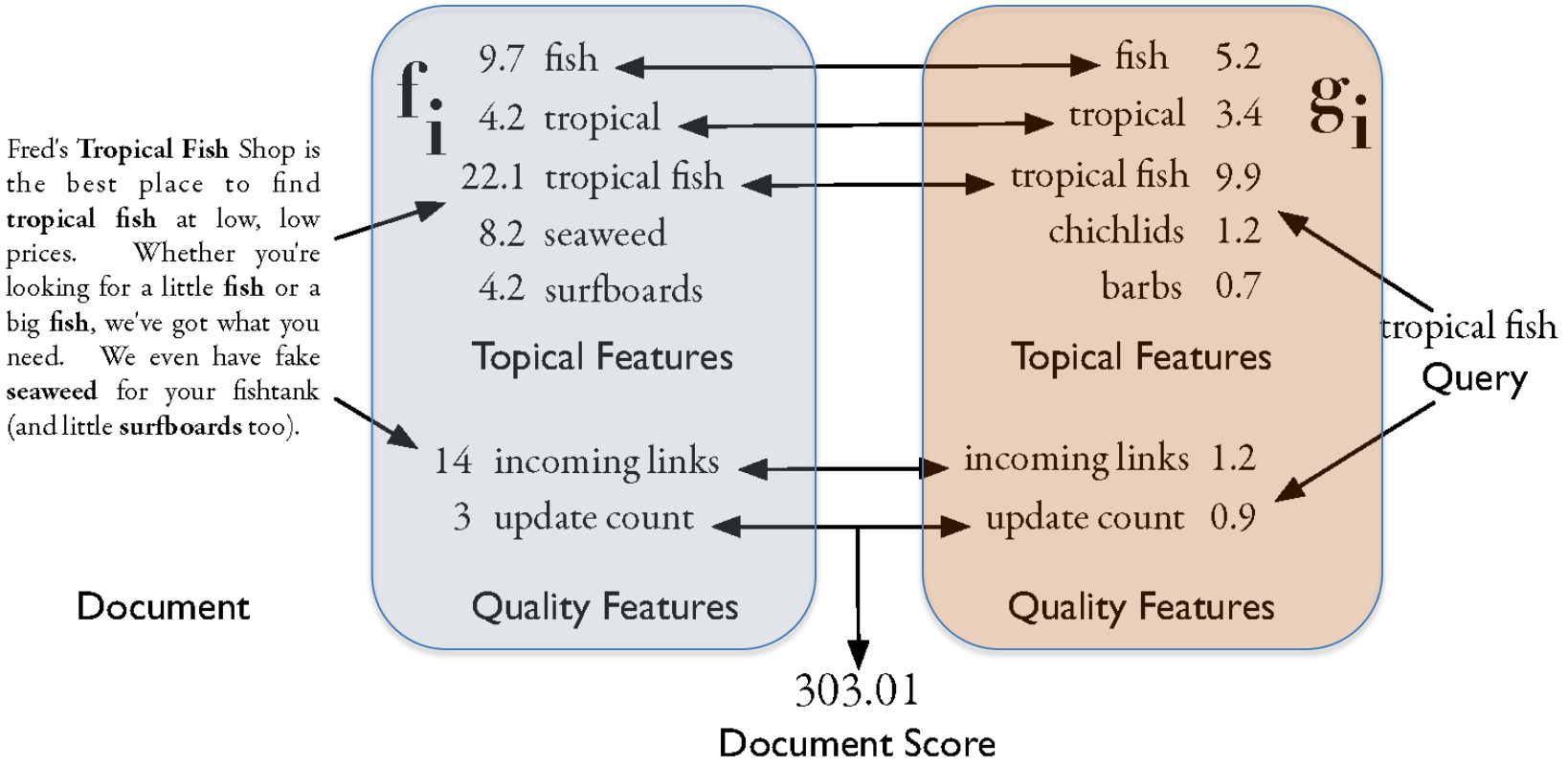


More Concrete Ranking Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

summed over all features i

f_i is a document feature function
g_i is a query feature function



INVERTED INDEXES

Hardware Basics behind Index Design

- Access to data in memory is ***much*** faster than access to data on disk.
  - Memory good for smaller, temporary storage
  - Does not scale to indexes for large collections
- Disk reads in 2 steps:
  - Disk seek: No data is transferred from disk.
  - Data transfer: usually block-based, reading/writing a block (8K to 256K or more) at a time.
- So: Moving a large chunk of data from disk faster than transferring many small chunks.

Inverted Index

- Each index term is associated with an *inverted list*
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

So:

  - Inverted file (inverted index): collection of inverted lists
  - Inverted list: collection of postings

    File → Inverted lists* → Postings*

Indexer steps: Token sequence

- Sequence of (Token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then by docID

Core indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Counts (frequency) information is added.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Another Example "Collection"

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

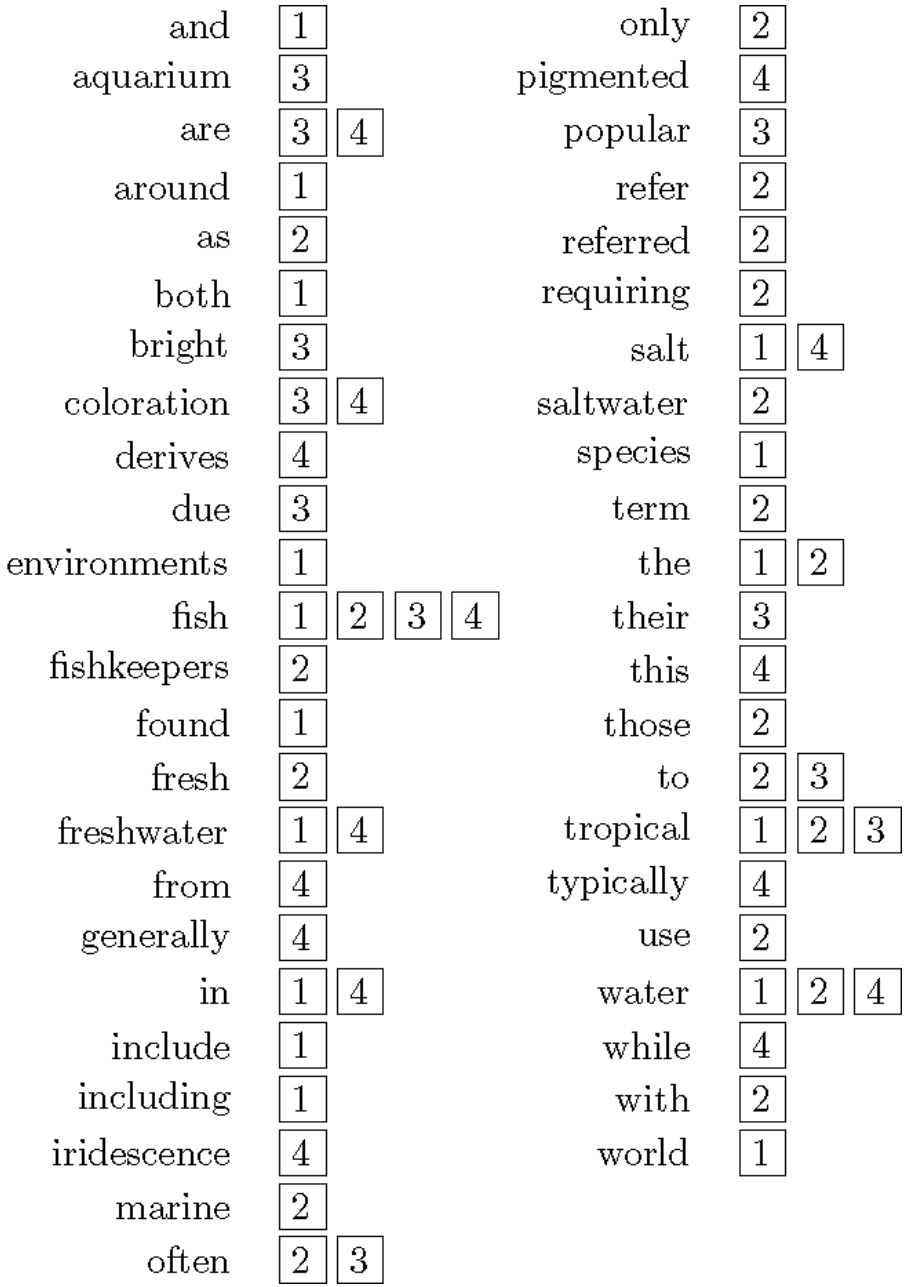
S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

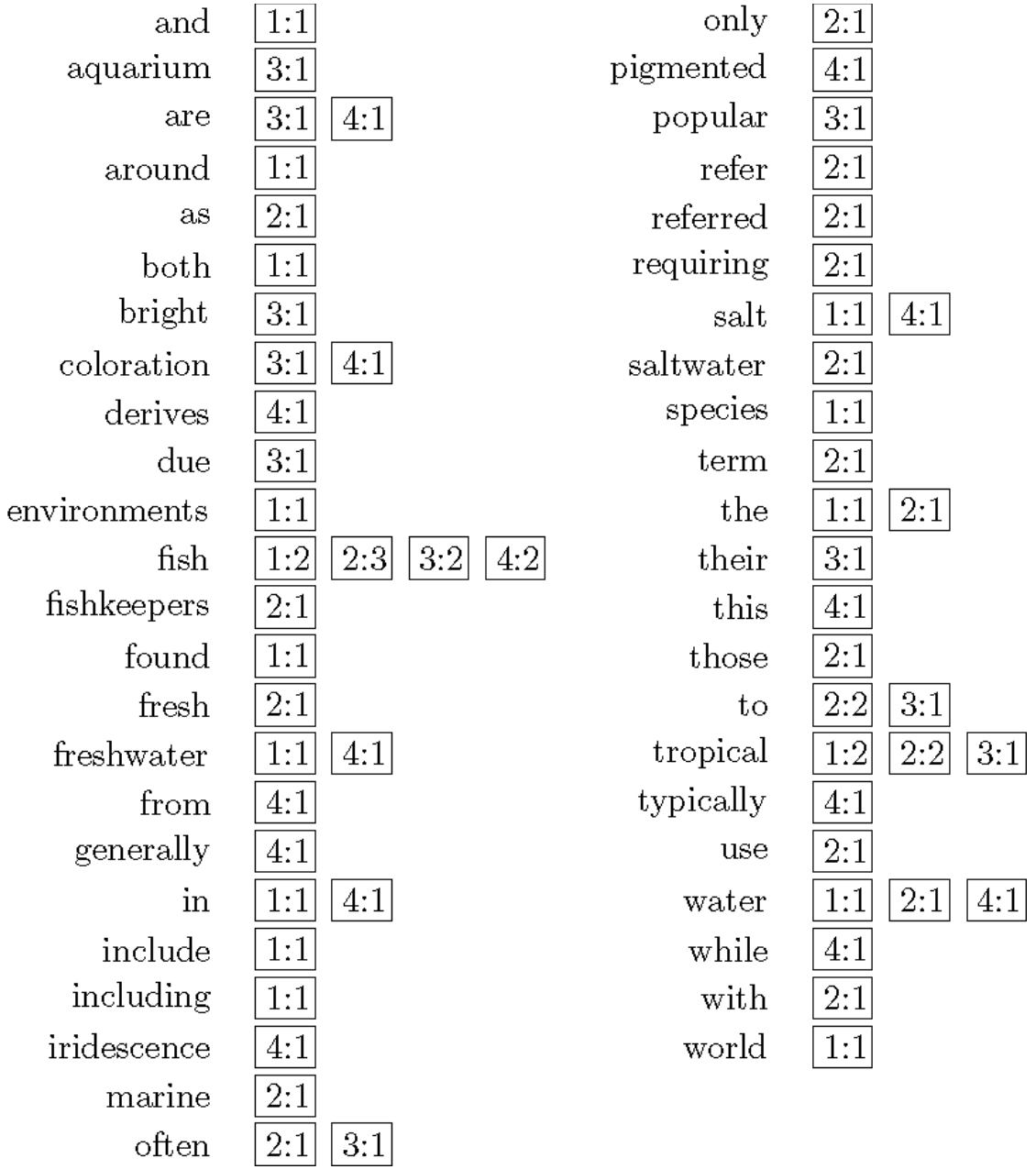
Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index



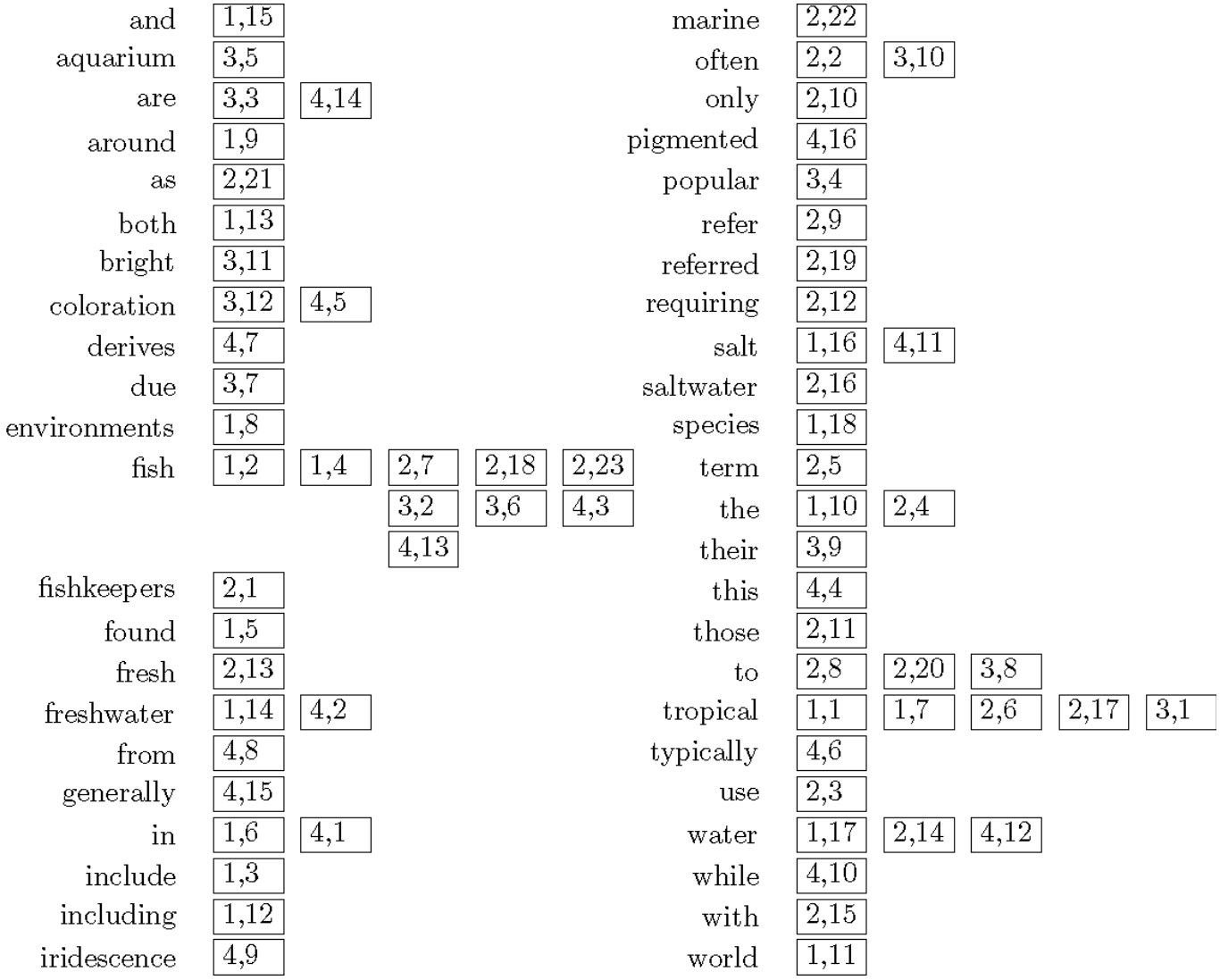
Inverted Index with counts

supports better ranking algorithms



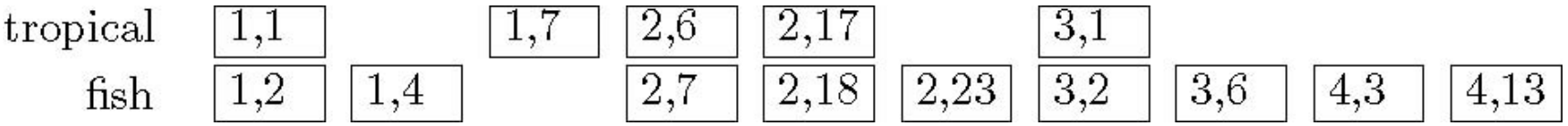
Inverted Index with positions

supports proximity matches



Proximity Matches

- Matching phrases or words within a window
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- Word positions in inverted lists make these types of query features efficient:

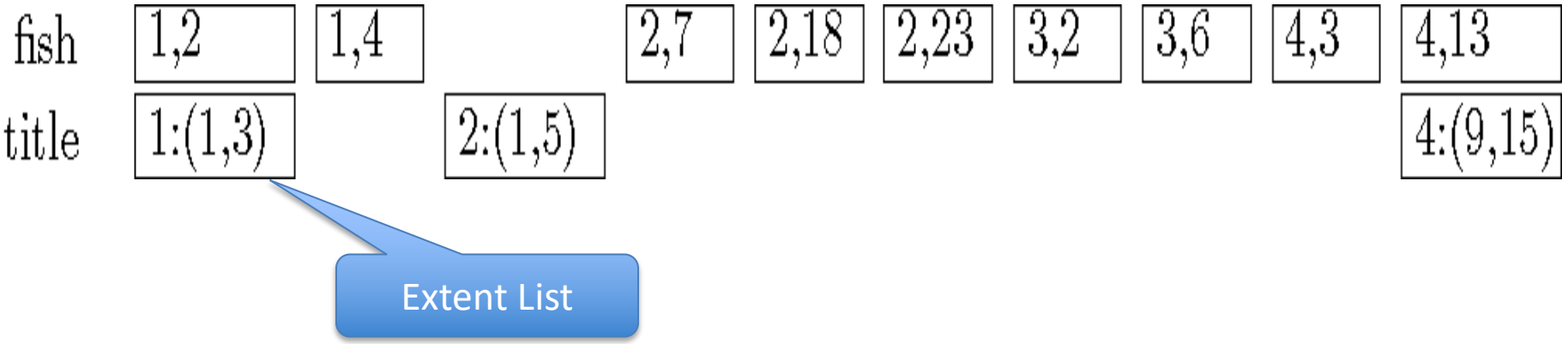


Fields and Extents

- Document structure is useful in search
  - *field* restrictions
    - e.g., date, from:, etc.
  - some fields more important
    - e.g., title
- Options:
  - separate inverted lists for each field type
  - add information about fields to postings
  - use *extent lists*

Extent Lists

- An *extent* is a contiguous region of a document
  - represent extents using word positions
  - inverted list records all extents (in all documents) *for a given field type*



Other Issues

- Could store precomputed scores in inverted list
  - e.g., list for “fish” [(1:3.6), (3:2.2)], where 3.6 is total feature value for document 1
  - improves speed but reduces flexibility
- Score-ordered lists
  - query processing engine can then focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries

COMPRESSING INVERTED INDEXES

Compression ..1

- Inverted lists are very large
  - e.g., 25-50% of collection for TREC collections (using Indri search engine)
  - Much higher if n-grams are indexed
- Compression of indexes saves disk/memory space
  - More of the index per server
  - But: typically must decompress lists to use them
  - Best compression techniques have good *compression ratios* **and** are easy to decompress
- *Lossless* compression – no information lost

Compression .. 2

Basic idea: Common data elements use short codes while uncommon data elements use longer codes

Example: coding numbers

- Number sequence: $0, 1, 0, 3, 0, 2, 0$
- Possible encoding: 00 01 00 10 00 11 00
- Encode 0 using a single 0: 0 01 0 10 0 11 0
- Only 10 bits (not 14), but... *Ambiguous* encoding
- **Another** way to decode: 0 01 01 0 0 11 0
- which represents: 0, 1, 1, 0, 0, 2, 0

Not binary code

Compression Example

- Instead, we can use unambiguous coding:

Number	Code
0	0
1	101
2	110
3	111

- which for: $0, 1, 0, 3, 0, 2, 0$

- gives the encoding: $0\ 101\ 0\ 111\ 0\ 110\ 0$

  (13 bits)

No other split possible.

Delta Encoding .. 1

- Word count data is good candidate for compression
  - many small numbers and few larger numbers
  - encode small numbers with small codes
- Document numbers are not very predictable
  - but differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*:
  - encoding differences between document numbers (*d-gaps*)

Delta Encoding .. 2

- Inverted list (without counts) for some word

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

- Differences between adjacent numbers

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

- Differences for a high-frequency word are easier to compress, because they're more likely to occur in doc-ids closer to each other e.g., $1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$

- But differences for a low-frequency word are large, e.g.,

$$109, 3766, 453, 1867, 992, \ldots$$

Bit-Aligned Codes

- Bit-aligned: Breaks between encoded numbers can occur after any bit position
- *Unary* code
  - Encode *k* by *k* 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

Unary and Binary Codes

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- Binary is more efficient for large numbers, but may be ambiguous

Elias-γ (Elias-Gamma) Code

- Use best of unary & binary
  To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$
    - k_d is number of binary **d**igits in k_r , encoded in unary
    - k_r **r**esidual, encoded in binary (in k_d bits)

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Elias-δ (Elias Delta) Code

- Elias-γ code uses no more bits than unary, many fewer for k > 2
  - 1023 takes 19 bits instead of 1024 bits using unary
  - In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits
- To improve coding of large numbers, use Elias-δ code
  - Instead of encoding k_d in unary, we encode $k_d + 1$ using Elias-γ
  - Takes approximately $2 \log_2 \log_2 k + \log_2 k$ bits

Elias-δ (Elias Delta) Code

- Split k_d into:
  - $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
  - $k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$ + 1
  - encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

```
#
# Generating Elias-gamma and Elias-delta codes in Python
#

import math

def unary_encode(n):
 return "1" * n + "0"

def binary_encode(n, width):
    r = ""
    for i in range(0,width):
     if ((1<<i) & n) > 0:
     r = "1" + r
     else:
     r = "0" + r
    return r

def gamma_encode(n):
    logn = int(math.log(n,2))
    return unary_encode( logn ) + " " + binary_encode(n, logn)

def delta_encode(n):
 logn = int(math.log(n,2))
if n == 1:
 return "0"
 else:
 loglog = int(math.log(logn+1,2))
 residual = logn+1 - int(math.pow(2, loglog))
        return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

if __name__ == "__main__":
    for n in [1,2,3, 6, 15,16,255,1023]:
        logn = int(math.log(n,2))
        loglogn = int(math.log(logn+1,2))
        print n, "d_r", logn
        print n, "d_dd", loglogn
        print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
        print n, "delta", delta_encode(n)
        #print n, "gamma", gamma_encode(n)
        #print n, "binary", binary_encode(n)
```

Byte-Aligned Codes

- Variable-length bit encodings can be a problem on processors that process bytes
- *v-byte* is a popular byte-aligned code
  - Like Unicode UTF-8
  - Variable byte length, hence, v-byte
- Shortest v-byte code is 1 byte
- Numbers are 1 to 4 bytes, with
  high bit 1 in the last byte,
  0 otherwise
  other 7 bits in each byte used to encode number

V-Byte Encoding

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

k	Binary Code	Hexadecimal
1	1 0000001	81
6	1 0000110	86
127	1 1111111	FF
128	0 0000001 1 0000000	01 80
130	0 0000001 1 0000010	01 82
20000	0 0000001 0 0011100 1 0100000	01 1C A0

V-Byte Encoder

```
public void encode( int[] input, ByteBuffer output ) {
    for( int i : input ) {
        while( i >= 128 ) {
            output.put( i & 0x7F );
            i >>>= 7;
        }
        output.put( i | 0x80 );
    }
}
```

V-Byte Decoder

```
public void decode( byte[] input, IntBuffer output ) {
    for( int i=0; i < input.length; i++ ) {
        int position = 0;
        int result = ((int)input[i] & 0x7F);

        while( (input[i] & 0x80) == 0 ) {
            i += 1;
            position += 1;
            int unsignedByte = ((int)input[i] & 0x7F);
            result |= (unsignedByte << (7*position));
        }

        output.put(result);
    }
}
```

Example: Compressing inverted lists

- Consider inverted list with positions:

$$(1, 2, [1, 7])(2, 3, [6, 17, 197])(3, 1, [1])$$

- Delta encode document numbers and positions:

$$(1, 2, [1, 6])(1, 3, [6, 11, 180])(1, 1, [1])$$

- Can drop parens, commas etc.. [why?]

  1 2 1 6 1 3 6 11 180 1 1 1

- Compress using v-byte:

  81 82 81 86 81 83 86 8B 01 B4 81 81 81 -- 13 bytes!

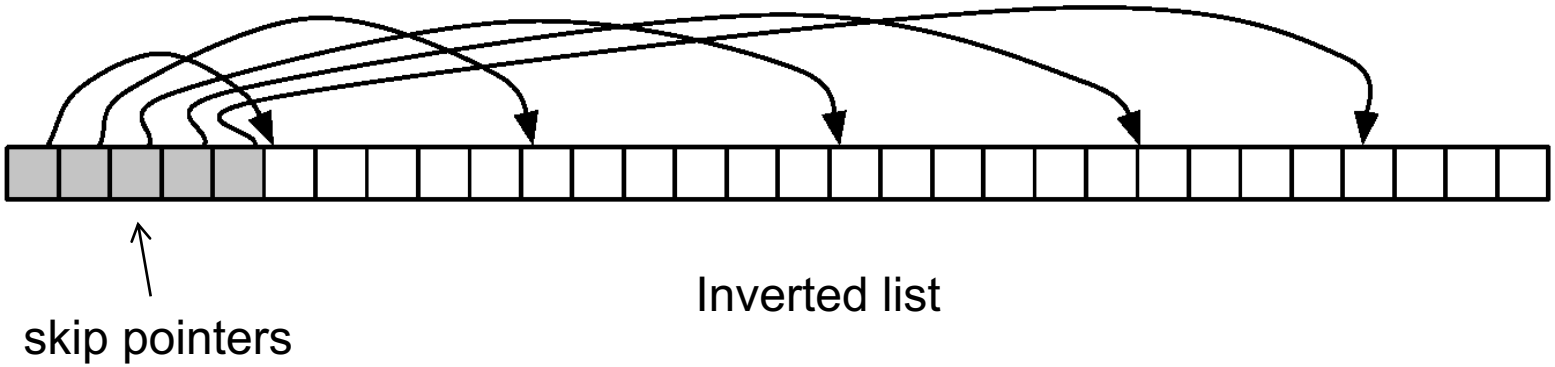
Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be very inefficient
  - "Skipping" ahead to check document numbers is much better
  - Compression makes this difficult
    - Variable size, only d-gaps stored
- Skip pointers: additional data structure to support skipping

Skip Pointers

A skip pointer (*d, p)* contains a document number *d* and a byte (or bit) position *p*

- Means there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*



Skip Pointers

Example

- Inverted list

$$5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119$$

- D-gaps

$$5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15$$

- Skip pointers

$$(17, 3), (34, 6), (45, 9), (52, 12), (89, 15), (101, 18)$$

… there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*…

Auxiliary Structures

- Inverted lists usually stored together in a single file for efficiency
  - *Inverted file*
- *Vocabulary* or *lexicon*
  - Contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - Either hash table in memory or B-tree for larger vocabularies
- Term statistics stored at start of inverted lists
- Collection statistics stored in separate file

CONSTRUCTING INDEXES

Index Construction

- Simple in-memory indexer

```
procedure BuildIndex(D)                          ▷ D is a set of text documents
    I ← HashTable()                              ▷ Inverted list storage
    n ← 0                                        ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                             ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

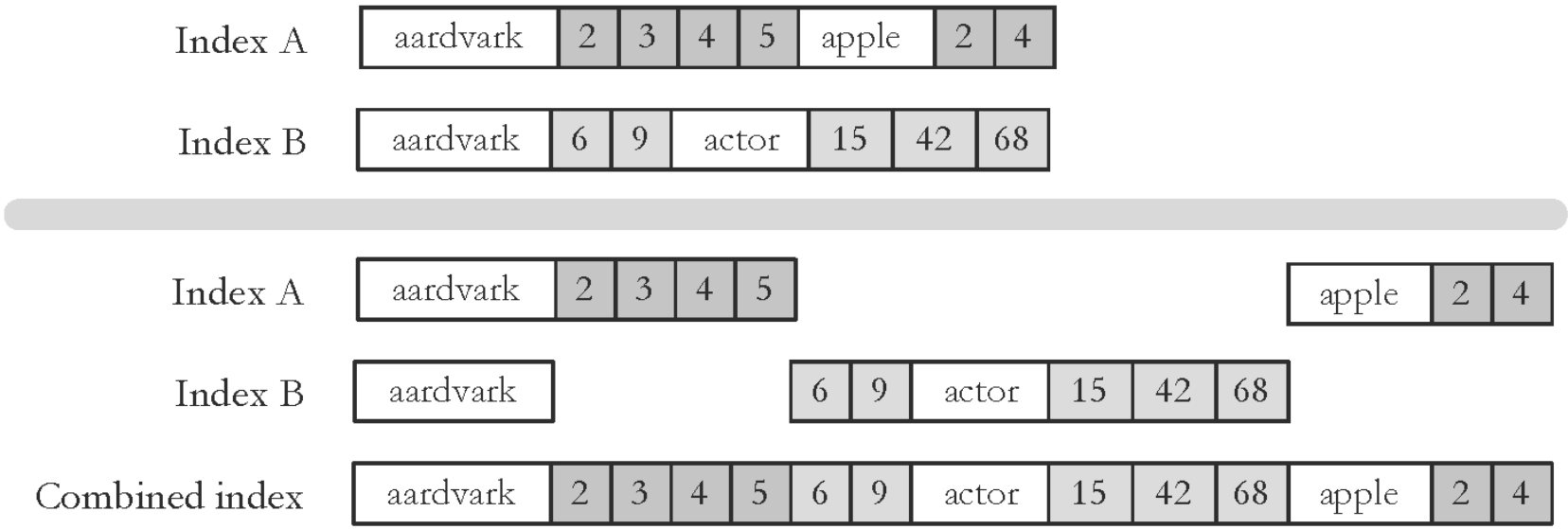
Create inverted list for token t if it's not already created

Add docid n to inverted list for token t

Merging

- Merging addresses: limited memory problem
  - Build the inverted list structure until memory runs out
  - Then write the partial index to disk, start making a new one
  - At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists must be designed so they can be merged in small pieces
  - e.g., by storing in alphabetical order

Merging


Index A
aardvark 2 3 4 5 apple 2 4
Index B
aardvark 6 9 actor 15 42 68
Index A
aardvark 2 3 4 5
apple 2 4
Index B
aardvark
6 9 actor 15 42 68
Combined index
aardvark 2 3 4 5 6 9 actor 15 42 68 apple 2 4

Distributed Indexing

- Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)
- Large numbers of inexpensive servers used rather than larger, more expensive machines
- *MapReduce* is a distributed programming tool designed for indexing and analysis tasks

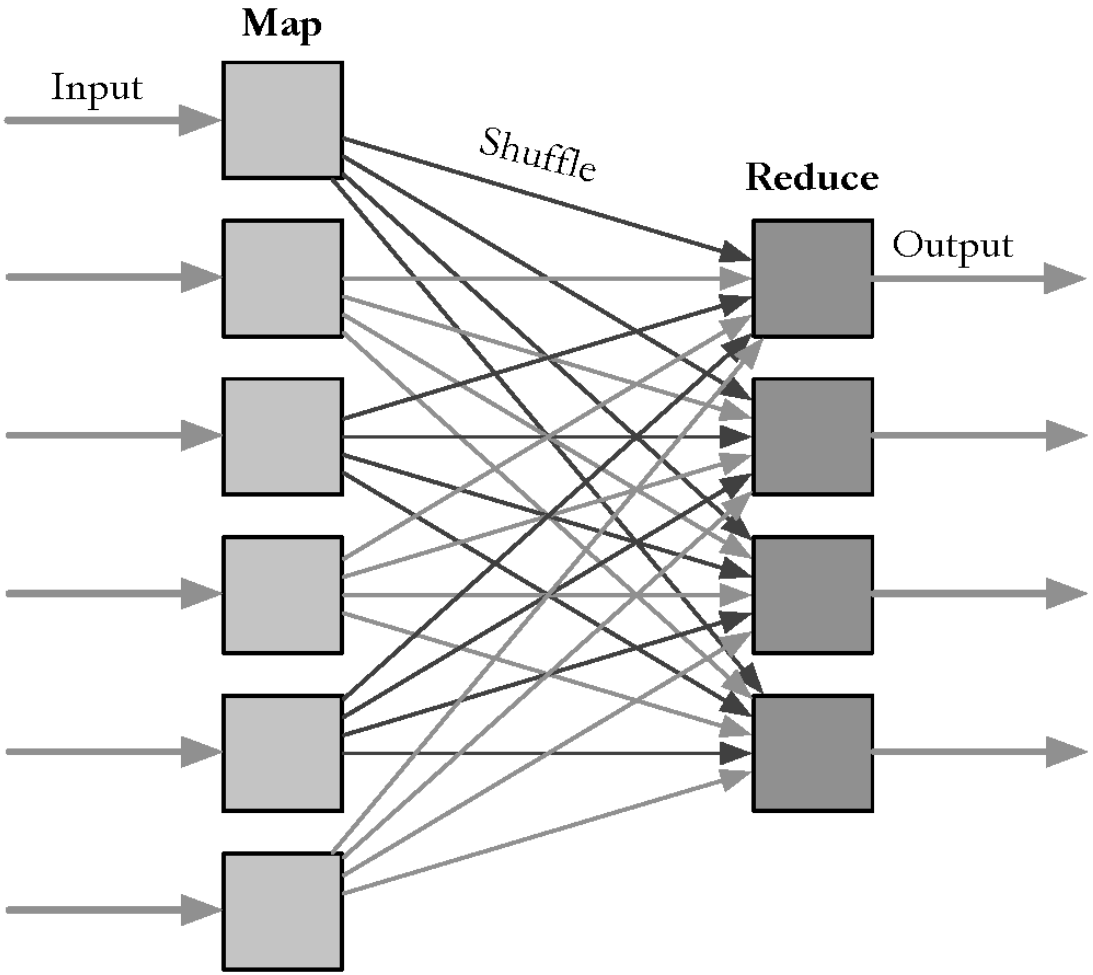
MapReduce: Credit Card Example ..1

- Given a large text file that contains data about credit card transactions
  - Each line of the file contains a credit card number and an amount of money
  - Goal: Determine the number of unique credit card numbers and total amount owed on each
- Could use hash table – but memory problems
  - counting is simple with sorted file
- Similar with distributed approach
  - sorting and placement are crucial

MapReduce ..1

- Distributed programming framework that focuses on data placement and distribution
- *Mapper*
  - Generally, transforms a list of items into another list of items of the same length
- *Reducer*
  - Transforms a list of items into a single item
- Definitions not so strict in terms of number of outputs
- Many mapper and reducer tasks run on a cluster of machines

MapReduce ..2



MapReduce ..3

- Basic process
  - *Map* stage which transforms data records into pairs, each with a key and a value
  - *Shuffle* uses a hash function so that all pairs with the same key end up next to each other and on the same machine
  - *Reduce* stage processes records in batches, where all pairs with the same key are processed at the same time
- *Idempotence* of Mapper and Reducer provides fault tolerance
  - *multiple operations on same input gives same output*

MapReduce: Credit Card Example ..2

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure
```

```
procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
```

Schema: index construction in MapReduce

- **Schema of map and reduce functions**
- map: input → list(k, v) reduce: (k,list(v)) → output
- **Instantiation of the schema for index construction**
- map: collection → list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

Example for index construction

- Documents
  - d1 : Caesar came, Caesar conquered.
  - d2 : Caesar died.
- Map:

  < Caesar,d1>, <came,d1>, < Caesar,d1>, < conquered, d1>,
  < Caesar, d2>, <died,d2>
- Reduce:

  (< Caesar,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, < conquered,(d1)>) →

  (< Caesar,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, < conquered,(d1:1)>)

MapReduce: Indexing Example

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, document:position)
            position = position + 1
        end for
    end while
end procedure
```

```
procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```

Updates? Index & Result Merging

- Index merging is a good strategy for handling updates when they come in large batches
- For small updates this is very inefficient
  - instead, create separate index for new documents, merge *results* from both searches
  - could be in-memory, fast to update and search
- Deletions handled using *delete list*
  - Modifications done by putting old version on delete list, adding new version to new documents index

multi data-center option

Summary

- Looked at reasons to index, and at inverted indexes.
- Studied compression, because we want to keep indexes smaller per document
- Looked at ways to construct inverted indexes

- Next week: **Query Processing**

Readings

- Croft, Metzler & Strohman (CMS), Chapter 5, Sections 5.1 – 5.6
- Optional: Manning, Raghavan & Schütze (MRS), Chapters 2, 4
  You can skip Section 2.4, 4.5

Questions?